Technical Report
CMU/SEI-87-TR-11
ESD-TR-87-14

(2)

Carnegie-Mellon University
Software Engineering Institute

F19628SC0003

# A Classification Scheme for Software Development Methods

Robert Firth
Bill Wood
Rich Pethia
Lauren Roberts
Vicky Mosley
Tom Dolce

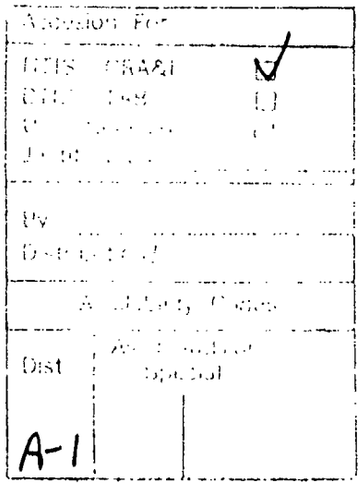
November 1987

**Technical Report**
**CMU/SEI-87-TR-41**
**ESD-TR-87-204**
**November 1987**

# A Classification Scheme for Software Development Methods

**Robert Firth**
**Bill Wood**
**Rich Pethia**
**Lauren Roberts**
**Vicky Mosley**
**Tom Dolce**

Real-Time Methodologies Project






**Software Engineering Institute**
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

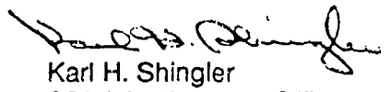
SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

**Review and Approval**

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

# Table of Contents

# A Classification Scheme for Software Development Methods

**Abstract.** Software development methods are used to assist with the process of designing software for real-time systems. Many such methods have come into practice over the last decade, and new methods are emerging. These new methods are more powerful than the old ones, especially with regard to real-time aspects of the software. This report describes a classification scheme for software development methods, includes descriptions of the major characteristics of such methods, and contains some words of advice on choosing and applying such methods.

## 1. Introduction

A major challenge in any engineering endeavor is taking a poorly structured, ambiguous, inconsistent, incomplete, and oversimplified requirements specification and turning it into a well-structured design. It is especially difficult in software engineering, a new field with few standards and procedures to act as guidelines for designers.

This report is one of a series concerning the classification, assessment, and evaluation of software development methods and tools. Its purpose is to describe a classification scheme for software development methods and provide relevant background material about these methods. In a second report, the assessment criteria for software development methods are developed, and a third report describes both a classification scheme for tools and evaluation criteria for selecting appropriate ones. Future reports will apply the classification scheme and assessment criteria to particular methods and tools.

This report is directed to three distinct audiences: those who write software requirements, those who design and implement software systems, and those who build tools to automate development methods. The people whom we expect to derive the most from this report, however, are those designers and implementors whose task is to choose methods and supporting tools for use in a specific application domain.

Since one of the stated goals of the Real-Time Methodologies Project is to look at methods and tools associated with real-time systems, this report begins with a context chapter, which first describes some important characteristics of real-time systems and then goes on to describe some of the conceptual issues that motivated us in this endeavor. Chapter 3 discusses the classification scheme for software development methods. Chapter 4 describes some of the issues in the development process, Chapter 5 provides a background on methods and their characteristics, and Chapter 6 discusses some of the issues relating to automated support of the methods. Finally, the last two chapters describe how to choose and apply methods.

# 2. Context

When people use the term **software design method**, they often have a particular aspect of the total design in mind. Our definition of this term is a broad one: start with a rather sparse set of requirements gathered into a requirement specification and derive a validated implementation of these requirements. The emphasis, however, is on the development of the target software, with minor references to the additional software needed for successful testing, installation, trouble-shooting, and quality assurance of the target software. One of the drawbacks of discussing software development as an abstract activity is that it tends to give a mechanistic view of the design process, while most practitioners would emphasize the contribution of individual designers [Brooks 87]. We agree with this position, and we reference the ideas of such practitioners to balance the more simplistic parts of this study.

## 2.1. Characteristics of Real-Time Systems

Before discussing methods of design, it seems judicious first to lay out some characteristics of real-time systems. This is not meant to be a complete list of such characteristics, but rather a short list of the major aspects that must be considered. Uncontroversial characteristics such as maintainability and quality, which apply to all systems, are not listed.

1. The major characteristic of real-time systems is that many different, interacting functions have severe **performance requirements**, especially in response time to events. Other functions monitor and control continuous processes, have strict periodicity requirements, and need specified computational resources within strict time intervals. These requirements are exacerbated by the need to respond to sudden "bursts" of events on top of the regular processing capability.

2. Real-time systems interact with various **sensors** and **emitters**, which transmit to the computer system signals representing the state of the environment and convey control signals to effect the environment. These devices may input analog measurements (temperature, speed, flow-rate), output analog values (setpoints), input digital measurements (on/off status of devices), or output digital signals (open/close switches).

3. Real-time systems are almost always **resource constrained**. In some cases the systems are CPU bound; in others main memory allocation is the problem; and in yet others the bottleneck occurs in communications between distributed components, access to shared data, or access to secondary storage. Unfortunately, concern with one resource (such as CPU) often leads to an implementation in which another resource (such as memory) becomes a severe bottleneck, necessitating large redesign efforts.

4. Real-time systems are **event driven**, and large parts of their behavior can be described as responses to events in the environment. Capturing the interaction of the system with the environment is itself an important aspect of the design. In real-time systems, the behavior of these responses to different events must be synchronized.

5. Real-time systems often control and monitor systems that are continuously in operation and cannot afford downtime due to failure conditions; hence, **fault tolerance** is required. This has important software design consequences for recovery of functions without loss of time or critical information.

6. Some real-time systems must be in operation continually. Software may have to be

installed on a **standby** system, with an enforced **switchover** of control to the standby when the change has been installed. The system which was previously in control can then be upgraded and placed into operation in a backup mode.

7. Real-time systems are difficult to debug in the target environment because the delivered system may have residual flaws that have not been detected. Often the direct symptoms from these flaws are too obtuse to trace and correct. Hence the software design should include some tailorable **data extraction** capability to yield sufficient secondary symptoms to isolate and correct flaws.

8. Because real-time systems are usually installed into an operating environment different from the development environment, a set of problems can arise during **installation**. The software design has to allow for easy installation of the system and for procedures to validate the operational capability of the installed system.

9. Software is such a major part of the current generation of real-time systems that the interactions between the software, hardware, and the human operator need to be thoroughly inspected for failure modes that can lead to hazardous operating conditions. Thus, some care must be taken for **software safety** conditions [Leveson 83].

10. Department of Defense (DoD) real-time systems operate in an **evolving environment**. Software changes are necessitated by changes to the threat environment, to operational procedures, and to hardware. The design method should include an evaluation of the most likely ways the environment will change and also encourage a software partitioning strategy which can best accommodate such changes [Parnas 86].

There is a wide range of real-time systems, varying from firmware in a small chip with limited input/output and no persistent data, to systems supporting all battlefleet operations. Some are control-flow oriented, others have a large number of varied persistent objects, and still others are dataflow oriented. In very large systems, portions of the system will contain components with all of the above characteristics. There is probably no single method which is "the best" for the complete range of real-time systems.

## 2.2. Conceptual Issues

In deriving a classification scheme for software design methods, we addressed some of the conceptual issues described below.

1. How well does the method expose the flaws in the requirements and help the designer create a consistent and correct specification? How strictly can the method validate the specified functionality and behavior at the early stages of representation? How does the method handle the performance and resource constraints specified?

2. What is the relationship between the life-cycle models of software design, the different stages of representations of the software, and the activities performed by the designer?

3. What is the interaction between different representations of the software, and how can we use the higher-level representation to help derive the lower-level one? If there is minimal carry-over between levels, is it worthwhile going into great detail at the higher level?

4. At what points in the design process are human intelligence, judgment, and decision making most important, and why? What heuristics do good designers use when developing a design?

5. How can a design at one level be communicated to audiences with different roles to play in the development? The design is usually produced by a small group of people and used by a much larger audience; therefore, there should be a "reader-oriented" representation. What aspects of design are particularly difficult to deal with?

6. How can we represent the design so that changes defined at a high level can be easily incorporated through all levels and eventually be incorporated into the target system all in a disciplined, uncomplicated manner?

7. How can the design method best build upon previous work done by others, including reuse and salvaging of parts at all stages of representation?

8. Where do tools play their most useful role?

The intent of this project is to look at well-established and supported design methods. There are many such methods currently in use, and, in some cases, there are many commercial tool sets available that automate some aspects of a single method (for example, structured analysis). Some vendors are also extending the capability of the original methods in their automated tool sets (for example, incorporating state transition diagrams with structured analysis). However, there is also a significant body of research that examines certain aspects of software design, and we cannot help but be influenced by this research, especially in the area of developing criteria for good methods. The most relevant research areas are those of formal methods and artificial intelligence. The work on formal methods is oriented toward specification of the design using mathematical representations such as CSP [Hoare 85], [Hayes 87], VDM [Bjorner 82], or specification languages with an underlying theory, such as Larch [Guttag 85]. The work on artificial intelligence is oriented toward automatic programming [Balzer 85], wide-spectrum languages [Smith 85], and transformations [Partsch 83]. While these and other efforts are undoubtedly important research developments, evaluating their capabilities is not the major concern of this project.

# 3. Classification of Methods

In this chapter, we discuss classification of design methods. We introduce the classification scheme here to lend a focus to the remainder of the report and to explain why certain issues are given only cursory attention. We first introduce the two independent axes of the classification. Then we discuss the classification scheme. More detailed historical perspectives of methods are included in later chapters.

## 3.1. Software Development Stages

The development stages are listed below. Each development stage is characterized by the aspects of the system that the stage represents and the way those aspects are represented.

The **requirement** is a description of what the end-user audiences view as their needs and is often a rather eclectic description. It usually covers the needs of each audience in the end-user community in an uneven manner, with some aspects (such as ease of installation) often overlooked. It often describes some functions (such as a scheduling mechanism) in very general terms, but others (such as a communications protocol), extremely thoroughly. Often, important requirements are not addressed in the requirements specification, and they need to be exposed and handled as they arise. The earlier these issues are addressed and resolved, the better the prospect of delivering a high-quality product within budget and schedule.

1. The first step is to take the ambiguous, incomplete, and inconsistent requirement and turn it into an almost flawless **specification**. The specification describes what the software is to do and the constraints to be imposed on the designers. Although the design process is not the primary consideration in this paper, it is worthwhile noting that production of the specification is not limited to a front-end activity, but will proceed throughout the life cycle of the system.

2. The **design** representation describes how the system is structured to satisfy the specification. It describes the system in a large-grained manner and defines the breakup of the system into major tasks. It describes persistent data objects and their access mechanisms, the important abstract data types and their encapsulation in the tasks, and the message structures between the tasks. There must also be some consideration for allocating resources and satisfying the performance requirements.

3. The final development stage is **implementation** with source code, object code, resource usage, and initialized data structures. This is the level at which algorithms are developed and represented explicitly.

## 3.2. Views of the System

Various views of the system are needed to describe its intended and actual operation [Harel 86]. These views are relevant at each stage of the system development and are enumerated below.

1. The **functional** view shows the system as a set of entities performing relevant tasks. This view includes a description of the task performed by each entity and the interaction of the entity with other entities and with the environment. The functional view is often the starting point for the design process, since it is commonly the way the system is decomposed into smaller and simpler parts.

2. The **structural** view shows how the system is put together: the components, the interfaces, and the flow between them. This view also shows the environment and its interfaces, and information flows between it and the system. Ideally, the structural view should be an elaboration of the functional view. Each entity in the latter view is decomposed into a set of primitive software components that can be implemented separately and then combined to build the entity. The design process, therefore, generally converts a functional view into a structural view. However, the structure of a system is influenced by resource constraints which prevent the use of arbitrarily many or arbitrarily large components. The structure is also influenced by certain implementation constraints that require the use of specific types of component (e.g., Mil-Std-1750a processors) or require that components be connected in a specific manner (e.g., by Mil-Std-1553 buses). The structural view should include a definition of the number and dimensions of entities to allow for resource estimates.

3. The **behavioral** view shows the way the system will respond to specific inputs: what states it will adopt, what outputs it will produce, what boundary conditions exist on the validity of inputs and states. This includes a description of the environment that produces the inputs and consumes the outputs. It also includes constraints on performance that are imposed by the environment and function of the system. Real-time systems, especially, have performance requirements as an essential part of their correct behavior. The behavioral view should include a definition of the expected workload and the required responses of the system to this workload.

## 3.3. Classification Scheme

The classification scheme shown in Table 3-1 uses the system stages (specification, design, implementation) on one axis, and the views of the system (behavioral, functional, and structural) along the other axis. The requirements analysis stage is **not** included, since it is informal, and we believe there is little to be gained by including it. A method will be classified by marking the appropriate box or boxes. A later report will further detail this classification scheme.

| Views of the System | Specification | Design | Implementation |
|---|---|---|---|
| Functional | | | |
| Structural | | | |
| Behavioral | | | |
| | | | |

**Table 3-1:** Stages of Development

# 4. Development Process

To describe development methods, we must also describe some related development process issues. This chapter outlines the issues we believe are important in the development process and includes discussions on systems engineering, system and software life cycles, development paradigms, and management.

## 4.1. System Design Phases

There are three phases in the development of large military systems as described in DOD-STD 2167 — concept exploration, demonstration and validation, and full-scale development. A short summary of each is given below.

1. The **concept exploration** phase (CE) is a study of the technical feasibility of the system in which each contributing technology (and science) is examined to determine how it is applicable to the system being considered and to advance the technologies as necessary to make the system feasible.
2. During the **demonstration and validation** phase (DEMVAL), the applicable technologies are integrated in a loosely coupled manner to demonstrate the feasibility of the system. The demonstration is not necessarily a full-scale model, but may demonstrate that critical parts of the system work separately and that the technology to integrate them exists.
3. During the **full-scale development** phase (FSD), design systems are built by building upon the appropriate technologies from the DEMVAL phase.

The software developers should use a method at each phase appropriate for that phase. For example, software developed for the concept exploration (CE) phase will not have to be supported over an extended period of time to the same extent as software in the FSD phase. Hence, a development technique lacking a maintenance emphasis may be appropriate for this phase, but is totally inappropriate for the FSD phase, in which maintenance is a primary concern.

Since the phases also have very different objectives and may be widely separated in time, it is unlikely that there is much direct carryover of software artifacts between phases, though one should not discard useful DEMVAL software which is applicable to FSD. This is especially true of software environments and tools which have been developed during DEMVAL and can be used directly in the development of FSD software. It is less likely that application software produced for the DEMVAL phase can be used directly in the FSD phase.

## 4.2. Systems Engineering

The field of **systems** engineering involves many topics, including design, installation, and operational aspects of a system, and the integration of different artifacts (hardware, operational software, software tools, test procedures, operational procedures, and maintenance procedures). The systems engineer has to understand diverse technologies and integrate these technologies into a well-designed system to meet end-users' expectations. Systems engineering is usually performed by a team of people with expertise in a sufficient number of the relevant areas to

consider the requirements and constraints associated with each technology of interest. They are supported by experts with more specialized knowledge in each technology area. A list of typical technology areas is given below.

- sensor technology
- human-machine interface
- computer processors and peripherals
- communications
- applications domains
- fault tolerance
- programming languages
- operating systems
- software engineering

One of the most difficult problems facing the systems engineer is that each technology is advancing at its own pace (while still being somewhat dependent on the other technologies), yet the systems engineer has to choose the stage of each technology at which they all can be successfully integrated. Thus the risk of having out-of-date technologies is balanced against that of not being able to produce the system on time and within budget. (Of course the software engineer faces the same problem, with a somewhat more restricted range of technologies.)

The systems engineer is often working from crude estimates of both the workload and resource requirements for each functional component of the system. These crude estimates are made from past experience and are greatly aided by reusing or salvaging both hardware and software (either the artifacts or the ideas). Of course the architecture that emerges is the result of balancing costs and risks. The chosen architecture will be perceived as one with a low risk of bottlenecks and a high potential for satisfying the functionality within the prescribed budget and schedule, while not gold-plating the solution.

One of the traditional aspects of the systems design is to specify what functions will be provided by hardware and what functions by software, as well as how the software modules will be allocated to the hardware components for execution. Systems engineering accounts for fault-tolerance in the hardware, describing how the system operation is affected by failure of hardware components. The systems design also defines how the communications between the software executing on different processors will be achieved. Systems engineering must, therefore, be based on good estimates of the software resource requirements in order to assign software to hardware components and to make sure that the capacity of the communications channels is sufficient to handle the data and signal traffic. This has always been a difficult task, and often the resources allocated to some software functions were insufficient, or the channel bandwidth was inadequate, or the storage capability of the media was inadequate, and so on. A few of the major drawbacks of such a design philosophy are listed below.

- The software design often reflects the hardware design, and small hardware changes can cause significant software changes.

- If bottlenecks arise, a reallocation of the software components to the hardware components is often necessary, hence changing the software design, which is dependent on the hardware configuration. An alternative solution is to include hardware with more power, but this often leads to acquiring new operating systems, necessitating a software redesign.

These problems with this approach to system engineering are inherent in emerging architectures such as PAVE PILLAR [PavePillar 85], which consists of both loosely coupled and tightly coupled processor clusters, with each cluster containing special-purpose processors, general-purpose processors, data communications channels, and signaling communications channels. The designer has more flexibility to assign software functions to processors, but the assignment is difficult to accomplish, and is often performed in the later stages of design and implementation.

## 4.3. Life-Cycle Issues

The software life cycle within the system full-scale development (FSD) phase involves managing and controlling the software acquisition through all phases from requirements analysis to deployment to subsequent maintenance. DOD-STD 2167 describes how this is to be done, and clarifies in its foreword, "The intent of the standard is to permit any systematic, well-documented, proven software development methodology. As such, this standard should be selectively applied and tailored to fit the unique characteristics of each software acquisition program, and the particular life-cycle phase in which it occurs." The life-cycle activities are well documented in the standard and in other reports and will not be described in detail here. Unfortunately, some overloading of the term *phase* is necessary in this report since both the system development (EC, DEMVAL, FSD) and the software development portion of FSD use the word phase. For the remainder of the report, *phase* can be taken to mean the phase of the software life cycle within FSD. We will specifically use "system phase" as necessary.

DOD-STD 2167 also details how the FSD is to proceed. The standard suggests that the systems design use functional decomposition to separate the system into software and hardware configuration items as a first step during the software requirements analysis phase of FSD. This development paradigm has significant problems, some of which are detailed in [Firesmith 87], and it is for many of those reasons that we chose to take an approach to software development methods independent of life cycle. Further reasons are listed below.

- We are considering development methods for software for real-time systems, and the emerging technology is that of configurations of both loosely coupled and tightly coupled processor clusters. Breaking down the software to hardware assignments at an early stage for this architecture takes away the software designer's flexibility to assign software to hardware in the later stages of the design cycle. The longer the designer can postpone such physical allocation decisions, the greater the chance of avoiding bottlenecks.
- There is a perceived need to describe the operation of the system, as opposed to the software components. Many specification techniques do not draw rigorous boundaries between hardware, software, and operational requirements at an early stage but concentrate on describing what the system must do.

We have chosen to distinguish between development stages and life-cycle phases, since it makes it easier to understand that the maintenance phase in the life cycle causes changes to the representations at all the various stages in the development process.

There is an ongoing evolution of software life-cycle models. The most commonly used model is the **waterfall model**, while other models such as the **spiral model** are currently being proposed to overcome some of the deficiencies of the waterfall model. The waterfall model [Royce 70] is based on the principle of completing each design stage before going to the next stage, although it does allow for feedback to correct flaws in previous stages as they are uncovered in later stages. The major problem with this approach is that its uniformity does not allow the designer to concentrate on high-risk issues at a level low enough to resolve them. The spiral model [Boehm 86] is an attempt to improve on the waterfall model by allowing designers to concentrate on tackling the design by narrowing it to the remaining high-risk issues, assuming that these issues can be resolved at a lower level, and then elaborating on the solution to incorporate more of the functionality. The spiral model continually spirals through certain activities in this fashion until the design is complete.

## 4.4. Development Paradigm

The previous sections made no mention of how one derives representations at each stage and how one moves from stage to stage. This process of going from stage to stage is one of the key issues in the design process and is also quite controversial. We give a brief description of each of the competing paradigms that have been used over the years.

The **bottom-up** paradigm is often used in two slightly different ways:

1. To denote that the designer leaps from requirements to implementation descriptions of how to solve problems, skipping the specification portion, and dealing only in a cursory manner with the design stage.
2. To denote that the designer identifies the low-level functions first, designs and implements them, and then proceeds to design and implement the next higher level, and so on.

The problem with either interpretation is that some important design considerations are embedded in the code, and if these are incorrect, they lie undetected until they are revealed during system integration.

The **top-down** design paradigm usually means that designers start with a single representation of the system, and then partition the system by some scheme (usually functional), describing the interfaces between the partitions as they go. This paradigm emphasizes the structural and functional viewpoints as opposed to the behavioral and leads to designs where the behavioral aspects are buried in the code.

In the **middle-out** paradigm, designers reject both top-down and bottom-up approaches and proceed by choosing some aspect of the system, specifying it thoroughly, perhaps making a first-pass design to determine its feasibility, and then concerning themselves with designing de-

tails of interfaces to the rest of the system. This paradigm has an added advantage: the designer has an early understanding of the performance, resource, and other problems associated with the design.

The **rapid prototyping** paradigm describes the building of an inexpensive prototype of portions of the system to validate the requirements before committing to an expensive full-scale production. This seems to be a good way of approaching both the high-risk portions of the system and the specification of user dialogs.

The **reuse** paradigm is currently the basis of extensive investigations. People reuse software in two principal ways:

1. Mathematical and statistical library functions are often reused without giving the subject much thought.
2. People build a new system by salvaging what they can from a known and similar previous effort. The closer the new system is to the previous effort, the more likely people are to reuse its worthwhile products.

The above, however, are limited examples of reuse, which is now seen as a major factor in increasing both the productivity of software developers and the quality of the software product.

An idealized paradigm is one in which the designers express all problems completely at the specification stage. They then use some well-defined procedures and heuristics, supported by powerful tools **and** their own judgment, to derive a high-level design to satisfy the specified functionality and behavior within the performance and resource constraints. Conventional wisdom has it that this is done in an iterative manner. Prototyping of some algorithms is necessary to obtain measurements for predicting operational performance and resource usage; recording the design in a top-down manner is desirable from the standpoint of reviewers and maintainers.

## 4.5. Management Issues

Software engineers produce a product to meet a set of users' needs and to be used by people other than the developers. As for any other such product, the definition, design, production, and maintenance require management of both the people developing the product and the process being used. Management issues are those covering costs, schedules, personnel assignments, productivity, and coordination and communication across organizational boundaries. The focus of this section is directed at the overlap of methods and management concerns.

Methods should support a **planning** process that begins with the development of a definition of the system to be built, allows for the early examination of a number of possible designs, and results in a detailed statement of what will be produced, when, how, and at what cost. To support this process, methods should lead to artifacts that can be understood and reviewed by the appropriate audiences. The methods should promote rapid development of high-level design representations that can be analyzed against a set of constraints, compared, and judged to determine the most feasible design approach. The methods should prescribe the generation of a sufficient number of intermediate products that are produced at a reasonable enough frequency to support

the creation of detailed project plans. Schedules and budgets presented in the plans should be based on these intermediate products.

Methods must integrate with the process of **organizing and staffing** a development project. Methods prescribe a way and sequence of doing things — what activities should be performed in what order. Requirements analysis and design methods, for example, often lead to an early partitioning [see Section 5.1.3] of a problem into manageable pieces. Projects should be organized so that they parallel the partitioning prescribed by the selected methods. Interrelationships between subsystems can be used to identify interrelationships between working groups. Particular characteristics of individual subsystems dictate that individuals with different combinations of skills may be required to further develop each subsystem. Methods are not substitutes for individuals' skills but can help identify which skills must be developed within, or acquired by, a particular organization

Methods should also support the process of **tracking** project progress and **controlling** ongoing activities. The methods should prescribe, for each of the intermediate products, a set of rules, metrics, or guidelines that can be used to judge the completeness and quality of the intermediate products. Review teams, using these rules in conjunction with their own knowledge, should be able to judge the status of each intermediate product. These judgments, along with an understanding of the relationship between intermediate products and the overall project, help assess project status, pinpoint areas of difficulty, **assess risk**, and focus on problem-correcting activities. This support is important in all areas of development, but plays an especially important role when the methods' intermediate products are deliverables in a contractual sense.

Finally, methods and management can be integrated to develop an understanding of the engineering process within a particular organization and help **direct** future activities. Use of methods on projects, including the activities of gathering and analyzing data on the use of the methods, allows an organization to understand its strengths and weaknesses. This information can help identify the key areas where skill building and focused management activities can have a large payoff.

## 4.6. Audiences

The artifacts associated with software development have many audiences, each with its own point of view and expertise. This means that, in principle, there ought to be various views of the representations available for perusal by each specialized audience.

The specification of what people do can be divided into those who actively create the representations, and those who examine the representations, with the recognition that some people do both, or perform different roles at different stages. The major audiences are outlined below.

1. **End-users** write the requirements for the system, and will install, operate, and maintain the system after it is delivered.
2. **Specification engineers** write the specifications from the requirements, resolving ambiguities, removing inconsistencies, and making sure that the specification is complete. They should specify what is to be done, not how it is to be done, although in some areas the constraints need to detail the implementation.

3. **Software designers** describe how the software is to be constructed to satisfy the specifications. This involves making optimization decisions on the best way to proceed given the constraints imposed in the specification. Examples of such constraints are performance requirements, resources available, and fault-tolerance capabilities. These constraints often influence the design as much as the functionality and behavior of the system.

4. **Algorithm developers** have specialized knowledge of a particular engineering discipline.

5. **Implementors** take the designed products and produce implementations of these designs.

6. **Quality assurance** people make sure that the products do what they are supposed to do. They deal with the quality of the products and ways of managing and testing the product.

7. **Managers** are concerned with providing leadership roles, with controlling the budgets and schedules related to the project, ensuring that problems are recognized early and resolved, and dealing with the various personnel assignments and problems.

# 5. Methods

In general, a method is "a systematic procedure, technique, or mode of inquiry employed by or proper to a particular discipline or art" (Webster). When applied specifically to software engineering, it could be defined as **a systematic approach to providing a software solution.** The method should, ideally, cover all aspects of the problem and, in the context of software engineering, should lead from an initial (imperfect) set of requirements to a satisfactory implementation, passing systematically through the intermediate stages.

## 5.1. Characteristics of Methods

At all stages in the development process, a representation of the system must be created. It is important to understand how the method contributes to this process. The method of designing a system at any stage can be considered from three points of view, namely:

1. What is the **form of representation** of the artifacts?
2. How are these representations **derived**?
3. How are these representations **examined**?

Each of these considerations is discussed in the following sections. The descriptions are not given in detail, but sufficient information is supplied to give the flavor of the approach. The characteristics will be elaborated in a further report describing how to select a method.

### 5.1.1. Representational Forms of the Methods

Chapter 3 described the proposed classification scheme. The most important characteristics of the representations depend on the stage and view of that representation. Some common characteristics are listed below.

1. **Graphical** representations are much easier to comprehend and are used to convey the understanding of the specification, design, and implementation. They also assist in developing a design and are most useful if they are part of the design and not just explanatory. Graphical representations prevent developers from dealing with detail prematurely and allow reviewers to perceive patterns more easily. Several problems detract from this comprehension of graphical representations:

   - **clutter** arising from representing too much information
   - **complexity** of the connections between objects
   - difficulty of **maintaining** connections as changes arise

   The alternative **textual** representation has the advantage of being more detailed and expressive. In general, a mixture of graphical and textual information is desirable. The textual descriptions provide completeness, and the graphical representations provide understanding of the structures.

2. The degree of **formality** of a representation is also important. The more formal the representation, the easier it is to reason about the operation and decide on completeness and consistency. However, a formal representation can obscure the "big picture" by its concern for specifying details. The formal systems are also more difficult to learn.

3. A large system has to be **partitioned** into subsystems, with the interfaces between the subsystems well defined. The same partitioning scheme should be applicable to derive a multi-layered representation.

### 5.1.2. Deriving the Representations

A method must not only describe the representations produced but must give some guidelines for deriving them.

1. Designers tend to use **past experience** as their guideline for approaching a problem. They pick and choose from their experience and try to apply to the current problem their knowledge of "what works under which circumstances." Some examples of the activities they perform are summarized below.

    - If possible, they will **reuse components** from another system, since this is the most productive path to take. They will only do so, however, if they have access to these components, confidence in their quality, and accurate specifications of their functionality and cost.
    - If they cannot reuse components directly, they may **salvage** these components, picking them up and changing them to apply to their particular problem.
    - Sometimes they cannot reuse or salvage components, but can still reuse the design principles or concepts which were applied previously by reorienting them to the problems of the design at hand.
    - Sometimes they can peruse a previous design and determine why it is not appropriate for the current system design. This often leads them to develop characterizations for the new problem.

2. Most experienced designers have an "issues list," either on paper or in their minds, when they approach a problem. This list contains important issues that are often overlooked in the design of large systems. Their critique of a design (their own and others) is often based on their consideration of one or more of these issues. Examples of such issues are:

    - Have the failure conditions and recovery characteristics been considered adequately?
    - Have the performance considerations been adequately expressed?
    - Have maintainability considerations been covered?

3. The method must contain **clear concepts** and the **abstractions** necessary to describe these concepts. This is especially true at the high-level design stage. For example, the specification may detail many related but different objects with similar functionality and behavior. It is the designers' responsibility to consider these as specializations of some well-defined abstraction, since it is usually easier to deal with as few concepts as possible at the design level.

4. Design is an **incremental** activity during which the designer creates or assembles parts of the system, examines the quality of the artifacts produced, and **iteratively** changes the parts to add more functionality, improve the performance, improve the interfaces to other parts, etc.

5. The **transformation** of artifacts from one stage of representation to another is extremely important. A separate section on transformations is included later.

6. The representations must be able to **evolve** over time, since software evolves in

response to changing hardware, operational requirements, and added functionality. Evolution of software generally retains as much of the original as possible, extending existing representations rather than developing innovative approaches, and leaving some redundant information which may become useful later.

### 5.1.3. Development Activities

During the development process, the design, usually in a state of transition, is inconsistent, incomplete, and ambiguous; even its feasibility may be uncertain. The whole purpose of a design method is to assist the developers in driving the design toward a consistent, complete, and unambiguous representation that is feasible at the physical level. Design usually progresses by a series of activities. These activities are referenced throughout this report and are listed below for the reader's convenience.

1. **Partitioning** is the activity of breaking a large problem into a set of smaller problems while defining the interfaces in a clear enough manner to maintain control over the breakup.

2. **Narrowing** occurs when designers cannot address the complete partition initially. They ignore certain aspects of the specification until they have a design for a portion of the problem. Once they have the partial design, they use it as the basis for reintroducing the initially ignored aspects.

3. **Elaboration** occurs when designers add more functionality to the partial design. If, for example, designers ignored recovery issues on the first pass of a design, they would be elaborating on the design when they finally include the recovery requirements. Elaboration occurs at a new design layer and may cause changes to dependent layers already created in the original design.

4. **Refinement** is the process of including more detail in the design. For example, expanding a node in a data flow diagram is a process of refinement. This process can be substantially aided by using an already existing design to produce a formatted template including all currently available information. Refinement is the way that the designer usually adds new layers to the representation at the same stage.

5. **Generalization and specialization** are dual activities. Generalization takes many object types with similar characteristics and forms a generalization containing all of the common attributes, while leaving the specialized attributes as characteristics of the original object types. Specialization starts with a single object type and derives different object types. These are often associated with the so-called **"is-a"** type lattice structure, and are more powerful if the specialized objects can have **exceptions**. (For example, a penguin is-a bird that cannot fly.) These activities allow designers to describe objects and the behavior of operations involving those objects at the appropriate layer of abstraction.

6. **Pruning** is the process of removing functionality from the product being designed. This may be done if the design is becoming too elaborate, if performance requirements cannot be met, or if the overall project appears to be infeasible.

7. **Transformation** is the process of taking a design representation at one stage and transforming it into the representation at the next stage, for example, transforming a set of data flow diagrams into a set of executable processes and modules. In many cases, transformation cannot be done automatically but requires some additional activities by the designers.

8. **Composition** occurs when fragments with well-known attributes are synthesized to perform a function that can be represented at a higher stage. Composition involves

customizing an assembly of components. This is extremely important, since it forces designers to reuse objects with well-defined properties and to salvage and change objects that almost satisfy the requirements. The objective is to improve both productivity of the design and robustness of the assembly. This requires the the designer be aware of objects at a lower level of representation in order to compose them into a superior level.

9. **Examination** describes the ways of determining the validity of the design, and it is a generalization of the specific activities of review, inspection, static analysis, dynamic testing, and rapid prototype building. This includes analysis of representations at the same stages and at different stages.

10. **Enhancement** is the activity of making changes to an existing design to change its functionality, improve its performance, etc. Enhancing a design is an aggregation of the other activities.

### 5.1.4. Examining the Representations

The usefulness of a method is heavily dependent on the ways available to examine the representations for consistency, completeness, accuracy, and all other desirable characteristics. The sooner problems are detected, the more cheaply they can be fixed.

1. We can use **formal proofs** to verify a system completely, but this method is not a panacea since it is a difficult process, understood only by the initiated, and unlikely to be accepted by the end user without some other representations. Using formal proofs is not only the domain of the well trained, but it is also difficult and time-consuming. As engineers, we should expect to derive some **heuristics** from the formal techniques and apply these in practice.

2. There are many ways of **analyzing** the different forms of representation for completeness, consistency, complexity, and coherency. These analytic techniques have three basic variations:

   - analysis of the same representational forms at the same stage (though across many levels and partitions)
   - analysis across different representational forms at the same stage
   - analysis across different stages and different representational forms

   It is clear that methods supporting powerful analytic capabilities of all the above categories are preferred over methods with less substantial capabilities.

3. The ability to **animate** the representational form is considerably important, since it provides immediate responses to the "what if" questions concerning the response to specified operational conditions and allows the design to be improved interactively.

4. The ability to **execute** is, of course, taken for granted once code has been generated for the final system, and all testing is predicated on the execution of the coded system in its final environment. One of the major problems is that specification flaws which are detected in this manner can cause significant redesign and adversely affect the delivery schedule and budget for the system. For this reason, there have been attempts to create executable specifications such as **PAISLey** (Zave 86). An alternative method is to produce executable **rapid prototypes** of high-risk keystone portions of the system.

5. The examinations should be **incremental** to allow the designer to examine portions on a stand-alone basis.

6. The ability to determine the **quality** of software using some predefined **metrics** is extremely desirable. The metrics used should be sufficiently powerful to allow the designer to select between alternative designs, and sufficiently simple to do so at a high level of abstraction.

## 5.2. Historical Perspectives

The purpose of this section is to give a historical perspective of specification and design methods without describing details of individual methods. All of the methods referenced are listed in Appendix A for convenience and are referred to below only by acronym.

The development and evolution of methods for specification and design are interwoven with the changes to related technologies that have occurred over the same time period. To give a chronological description of these developments is an imposing task. We have chosen just to describe the major influences.

1. The initial motivations for developing specification and design methods were both administrative and technical. From an **administrative** point of view, the methods broke the systems into individual pieces at different phases, stages, or levels. Estimates of various types could then be made about the pieces; budgets and schedules could be generated, monitored, and maintained. From a **technical** point of view, the methods curbed a tendency to go directly from requirements to coding by introducing more rigorous specification and high-level design mechanisms. One of the problems with the initial methods was that products were often deficient with respect to some technical aspect that was not well expressed by the method, for example, performance was unacceptable, or there were insufficient machine resources to support operation of the system during integration, or the user-interfaces were unwieldy. Such deficiencies often necessitated costly redesign, causing schedule and budget overruns. There has been a tendency recently to restore this balance by attempting to include more technical aspects into the methods while still improving the administrative aspects.

2. The emphasis on methods has come from the **data processing** community, and most of the early developments were specifically for that applications domain. Various methods were developed (JSD, SA/SD, SADT, PSL/PSA) that were applicable to the design of applications programs in that domain. These methods were initially developed in the mid-1970s and have been slowly evolving to incorporate new ideas and to respond to observed weaknesses in practice. One of the interesting developments over these same years, however, is that very good and powerful tool sets have been built to simplify the development of routine operational aspects from this applications domain. Examples of such tool sets are database management systems, fourth generation languages, and data dictionaries. These tool sets were not related to the design methods used previously to perform these functions using operating systems files and programming languages.

3. One of the major drawbacks of using these methods was the burden of tedious details to be checked by the designer. In actual fact, most designers were rather cavalier about those details, and just forced things to fit into the programming stages. The availability of workstations with bit-mapped displays, windowing systems, and access to laser printers has spawned much **automation** of the methods. Automation transfers much of the tedium from the designer to the machine and will undoubtedly improve the usability of the methods and the quality of the results. A more detailed discussion of automation is given in [Firth 87].

4. From the experience gained in using the methods over a number of projects and a number of years, the methods are being **improved** to remove some of their weak points. For example, although top-down descriptions may suit readers specifications, they are a poor process for creating a design. Many of the methods now recommend a middle-out approach to the design process, while maintaining the top-down description of the resulting design.

5. Many of the techniques developed for the data processing community were inappropriate for **real-time systems**, which are more oriented toward "response to events in the environment" and "periodic processes" than to transactions. Hence they require an effective way of describing this response and its interactions with other responses. There has been a tendency toward using methods that include behavioral descriptions at the specification stage, such as those described below:

   - extending established methods to incorporate state transition diagrams describing behavior [Ward 86]
   - creating new graphical methods incorporating multi-level state machines [Harel 86]
   - using formal mathematical descriptions to describe the behavior [Hoare 85].

6. The concept of **information hiding** was introduced in [Parnas 72]. It postulated that each software module should have direct access only to the bare minimum of information it needs for execution. Other necessary information should be obtained by accessing modules that understand the data structures. This led directly to the so-called A7 method. There was also some related work done on the Smalltalk environment [Goldberg 84], which used a software paradigm of treating objects as a data structure and a set of access mechanisms to that structure. These, and the work done on abstract data types [Aho 83], have led to the **object-oriented design** methods that are being used (especially with Ada as the target language, an example of which is described in [Booch 83a].

7. The original development methods were exclusively interested in development issues and had little involvement with the software **life cycle** or other **quality assurance** aspects of the software, such as testing and integration. This has changed dramatically, and there is an emphasis on quality assurance and life-cycle support included in many emerging methods (DCDS and TAGS).

## 5.3. Comparison and Classification of Methods

There are numerous surveys [Yau 86], classifications ( [Hesse 84], [Kelly 87], [McDonald 85]), and comparisons ( [DoD 82], [Bergland 81], [Floyd 86]) of development methods. Many of these papers propose classification schemes and evaluation criteria for methods. While we agree with most of the individual points made in these papers, have used them to develop our own classification scheme, and will lean heavily on them in developing our evaluation criteria, we feel that their proposals are not entirely appropriate for reasons such as those listed below.

- There was often no clear distinction between classifying methods (describing what methods do) and evaluating methods (saying how well they do it). We believe this distinction is essential.
- We believe that the classification scheme has to be simple (but meaningful), though the evaluation criteria should be extensive.
- The classification scheme often contained a curious mixture of technical, adminis-

trative, and economic judgments. We believe that the classification scheme should be technical, and administrative and economic considerations should be evaluative.

- In some cases, the distinctions were purely descriptive and would be difficult to use for comparing the capabilities of different methods. For example, describing one method as "data flow" and another as "data structure" does little to assist in classifying the methods.
- The reports were often selective in the methods chosen (limited to three to five methods), and hence there was less need to define general classification and evaluation criteria, since each technique could be probed for its strengths and weaknesses individually.
- In general, the papers failed to assist the reader in assessing the usefulness of one method over another.

# 6. Automated Support

This chapter describes some of the issues involved with automatic support for the software development process. The first subject discussed is tools which directly support the development methods; this is followed by a description of the capabilities which should be in the environments supporting these tools.

## 6.1. Tool Support of Methods

The notion of using tools in the production of software is not new. Editors, assemblers, compilers, linkers, and debuggers are examples of tools that are widely used to aid the soft are engineering process during the implementation phase of the life cycle. A classification scheme for such tools is described in a companion report [Firth 87]. However, most design methods can be directly supported by the tool functions listed below.

1. Most methods provide rules or heuristics for the construction of products and the relationships between them. Without automated tools, a designer must painstakingly go through all of the products by hand and perform the particular analysis to ensure that the guidelines are being strictly followed. An automated tool can check products (either while they are being produced or after completion, as appropriate) to ensure that the method is being applied correctly. The extent of **analysis** that can be performed on products is often tied to the formality of the notation. "Consistency checking as part of the tool is often limited due to the formality of the notation. In general, connectivity checks are performed, and identified items are stored in a so-called data dictionary. Through appropriate naming conventions, meaning about the identified objects is maintained" [Dart 87].

2. Compilers have been traditionally used to **transform** textual source objects written in a programming language into objects that will execute on a particular hardware and software suite. Ever since compilers have come into common usage, there have been dreams of extending this compiler paradigm to an earlier stage in the design cycle so that a design specification, written in a specification language, can be transformed by a "specification level" compiler into a programming language. There are many aspects to this subject, and a great deal of research has been done, as described in [Partsch 83] and [Balzer 85]. But there have been a limited number of successes. Although this work has yet to produce commercially viable products, it does give some objective criteria for evaluating capabilities. The work raises three different but related questions:

    - How can we best represent a specification?
    - How can we effectively transform that specification into a satisfactory design and implementation?
    - How can we easily make changes to the specification and derive a new design and implementation?

The state-of-the-practice is that transformations from specifications to design representations are performed by programmers. There has been a substantial body of work done to automate certain well-defined subsets of the design problem, such as database management systems (DBMS) [Date 86], form generators, screen painters, and report writers. These are the so-called fourth generation languages (4GLs), which would be more appropriately termed application generators

and are described in [Raghavan 86]. These have been developed both as general tools to be used with a particular commercial DBMS and as specialized tools associated with specialized applications.

After products are created by the application of the method, it may be necessary to **manipulate** (update or alter) them. With hard copies of the products, most often a new document must be created to replace an old, insufficient one. An automated tool can make copies of objects, thus facilitating alteration of an object, while still preserving the original product. Textual or graphical editors can support fast and efficient alteration of products. Furthermore, it is often helpful for the designer to view more than one product at a time. Sometimes the designer needs to trace through or view a group of products that are related in a particular way. An automated tool can enable the designer to view more than one tool output by windowing techniques which enable switching back and forth between a number of different references. Searching, sorting, and query facilities may also be provided to specify related objects.

The process of software development is cyclic in nature — often end products are achieved by iterations of the same process. Automated tools aid the task of iterating through a process by maintaining the results from previous processes, performing transformations, and providing editing capabilities. Using a tool, the designer can quickly explore different ideas that originated from the same starting point. In the case of user interfaces, a tool can display aspects of how the end-product software might behave. Some methods also contain formal executable languages. The execution of a language usually results in some form of analysis, for example, to point out discrepancies or incorrectness in the executable specification. The tool may also provide facilities for visible simulation, e.g., animated graphics, which can be a good means for communicating to the user.

## 6.2. Tool Consistency with Methods

The functionality and completeness of a tool must be weighed against the impact that its features have upon the application of the method. Following are tool features that can help or hurt a method:

- A tool can be used by a wide variety of organizations and users. If a tool can be **tailored** to user needs or to a particular user style, the tool has the potential to be used with more dexterity and at a faster rate than would be otherwise expected. When a tailorable tool supports a method, it is also important to understand the ramifications of tailoring, as it may affect the application of the underlying method. This not only may affect the results of applying the method, but may create considerable disparity among products of different tool users on team projects.
- The more **intelligent** a tool, the more functions it will perform without the user having to directly specify its initiation. In addition, the tool should be **helpful**, anticipating the user interaction and providing simple and efficient means for the execution of functions that the user requires. However, the tool should not automatically execute functions that may interfere with the design process and/or the application of the method or that may hinder experimentation by the user.
- A tool should be **predictable**. The user who is already familiar with or trained in the method the tool supports should not be surprised at any action or output of the tool

supporting the method. If the tool has omitted or changed an aspect of the method it supports, an experienced user of the method can become frustrated or confused, thus hindering productivity.

- The tool should be **flexible** and able to support the method enough to guide the user and ensure that the main concepts of the method are being adhered to. However, the tool should not be so rigid as to force the user to execute steps that might possibly be done at a later time or omitted completely.
- The tool should be **robust**, never generating incorrect transformations or executing a faulty analysis. In such cases, the tool seriously hinders the designer's work, as these mistakes are difficult to detect and locate.

## 6.3. Software Engineering Environment Issues

The environment provides capabilities to the user (designer or specifier in this context) and to the toolmaker. The environment has many characteristics which support both of these audiences and increase their productivity and the quality of the software they produce. Some example environments are listed below.

- The environments provided by specific operating systems such as UNIX and VMS.
- Programming language-dependent environments such as Cedar [Teitelbaum 81], Smalltalk-80 [Love 83], and structure-oriented environments such as GANDALF [Habermann 86], and Rational [Archer 86].
- Environments based on extending current operating systems and including database management concepts to control objects and relationships between objects. Examples are CAIS [MIL-STD-CAIS 85] and PCTE [Gallo 87].
- Environments which integrate the process of managing software products and the software engineers producing those products. Examples are DCDS [Alford 85], DSEE [Leblang 85], and ISTAR [Lehman 85].

It is not the purpose of this report to carefully inspect the issue of software development environments; however, the interested reader can refer to [Dart 87], [Houghton 87] for more information on the subject. It is our intention to identify what characteristics should be contained in the environment rather than in the tool set supporting a method. These characteristics are listed below.

1. The environment should provide a set of **common user interfaces** which the user can tailor individually and which give a common way of operating in the environment. These should include on-line manuals, "help" capabilities, and, wherever possible, iconic and graphic support.
2. The environment should allow for **portability** of tools between environments, and for the **interoperability** of objects produced by those tools between environments and between tool sets within an environment.
3. The environments should provide **object management** such that the objects can be stored and retrieved. The object management facility should support a rich relationship between objects and provide the user and the tools the capability to browse through the object base, to query the object base, and to produce reports about the contents of the object base. The objects in the object base should be heterogeneous, ranging from graphical representations to executable code. The object management should also include the efficient archiving of the object base, and, in a distributed system, should provide location transparency.

4. The environment should include **configuration management** to control versions, configurations, and releases of the objects.

5. The environment should also have some **process management** interfacing with the object and configuration management. Process management involves controlling how the software professionals are managed to produce the objects according to the overall objectives of the organization of which they are part.

6. The environment should gather data concerning the **workload** to which it is subjected and the work habits of the users (presumably while respecting their privacy).

7. The environment should provide mechanisms for **access authorization** to objects, including executable objects.

# 7. Using Methods

The software engineering process takes a requirements specification and constructs a satisfactory implementation in the manner previously described. The process is governed and directed by one or more methods that prescribe how the work is to be done, the stages of the development, and the actions appropriate to each stage. Although this process is usually presented as a smooth and orderly development, in practice the construction of real software rarely approaches this ideal. For example, if one part of the design seems to involve unusually high risk, it may be appropriate to build an early prototype of that part or conduct a brief feasibility study or performance estimation. The design method, however, might require that all parts of the design be brought to a certain stage before any implementation is done.

Another reason methods cannot always be applied in an ideal way is that the product must adhere to rigorous performance or size constraints. This may require the developer to be concerned with optimization issues at the design stage, while the approved method might require functional issues to be solved before performance issues can be addressed.

For these and other reasons, a method cannot be used inflexibly and without regard to the overall circumstances. Nevertheless, most methods do envisage an orderly progression from requirement to implementation.

## 7.1. Analyzing Requirements

The requirement is a description of what the user wants. It excludes many of the details which the user and supplier will eventually agree upon in the specification and serves as the basis for deriving the specification. Before proceeding to derive the specification, however, some analysis of the requirements is necessary, such as those listed below.

- identification of novel areas where proof of concept is needed
- identification of high-risk items
- prototyping of user interfaces
- selection of an appropriate set of methods for the remaining stages
- selection of environment and tools to support the development effort
- project planning, budgeting, and scheduling

The developer and user can be helped and guided by a method for requirements analysis that directs their attention to such issues. The method can also facilitate the planning and budgeting of tasks by identifying work items and dependencies at an early stage.

## 7.2. Deriving the Specification

We presume that the method of specifying the system has been determined previously, and in this stage we apply this method to the requirements and derive the appropriate specification. The specification states, as much as possible, what is to be done, not how it is to be done; it should include the entire requirement rather than the more limited automated portion of the functionality. The major initial activities to be performed are enumerated below.

1. The specification is often represented in a top-down manner, but it is rarely derived that way. The recommended ways of proceeding to derive the specification are the middle-out, object-oriented, information-hiding, event-driven techniques. As explained in [Parnas 86], the specification can still be represented to the readers in a top-down manner.

2. Identify reusable components which can be used in the system. These can be used as the basic building blocks around which the system is to be constructed. They can then be used as constraints on the specification derivation for the rest of the system. Reuse may also affect the requirements. It is often advantageous to change an unimportant requirement if doing so will allow for the reuse of available components.

3. Partition the problem into separable subsystems, each with its own specification team, and with the rules for interfaces between subsystems well defined.

4. Identify aspects which overlap subsystem boundaries, and separate them into one of the subsystems, defining interface mechanisms more strictly. This usually involves the process of generalization, is ongoing throughout the specification stage, and is especially important for some of the fundamental specification issues such as user interfaces, recovery, fault tolerance, and performance.

5. Develop the specification for each subsystem using the chosen method, remembering to account for the weaknesses of the method.

6. Concentrate on the specification of the previously identified high-risk items, and constantly review this list, update it, and reassign priorities as necessary.

7. Within each subsystem or portion thereof, specifiers usually start by narrowing the problem to one they can handle, refining the specification until they have a good grasp on the details, elaborating on the problem to include some of the ignored aspects, and continuing with this stepwise refinement until they have a completed specification. Every so often in this process, they have to remember to review the interfaces to the other subsystems and to pass judgment on high-risk items under their control.

Although the specification should be partitioned to make it most easily understandable to the various audiences, it is easier to maintain traceability between the specification and the design artifacts if they are partitioned in a similar manner. The method will usually control both the way the specification is partitioned and the way each component is represented. The form used by a method to represent systems and components is one of the characteristics that allow us to classify methods as discussed in Chapter 3.

## 7.3. Identifying the Components and Structure

The specification often models activities to include operations in the environment. The first step in the design process is to identify the automation boundary for each relevant portion of the system. The design method helps here by providing guidelines for how this boundary is to be drawn, procedures for specifying in a more formal manner the interface between the automated system and its environment, and the abstract behavior of the components external to the automated part.

Some design issues involve all subsystems and components, and these should be handled first. In particular, issues of robustness, fault tolerance, resource consumption, and performance involve the system in a holistic manner and cannot be cleanly abstracted into specific components or specific design stages. The method should help each subsystem designer remain aware of issues that affect all parts of the system.

It is desirable to use as much of the specification level description as possible, especially if it has been refined to a rather detailed level. There should be interactive tools to allow the designer to use the specification level to generate the design level, tools to check for consistency between both levels, and tools to analyze whether the design is likely to satisfy performance requirements.

There is a need to look at subsystems with differing but similar functionality and to generalize them if appropriate. It often happens that a specification defines several particular objects or functions that are similar; these may be specific examples of a general case, and the design method should recognize and exploit such generalities. This simplifies the construction of the system, since seemingly special components can be recognized as instances of a more general component that might already exist as a reusable artifact. It also helps with subsequent enhancement or maintenance, since the initial implementation offers more generality than originally called for, with no increase in complexity.

## 7.4. Implementing the Design

Once the components and their interfaces have been fully specified, it remains to implement them. Some methods will have so constrained and structured the design process that subsequent implementation is almost automatic: the design has been driven toward using a set of standard primitive components for which generic versions are already available, and toward interfaces whose definition can be constructed from a functional description. Implementation is then a matter of creating specific versions of the components, unit testing them, and integrating them into larger assemblies.

Other methods leave open many questions of implementation; for instance, issues of the physical representation of data types, the exact algorithms for data transducers, and the control hierarchy of the code components. In these circumstances, other rules must be used to guide the implementation. The basic problem of mapping real-world requirements into engineered functional artifacts is a hard one, one which most methods try to solve by restricting the design space toward paradigm solutions that are familiar and well understood. This fails when the problem is truly one for which only an innovative solution will work, but such problems are perhaps less common than usually claimed.

Most implementation methods specify an integration and test strategy. This strategy is usually designed to allow independent interface and component testing, bottom-up integration testing, and final system testing. In some cases, test data and scenarios will have been derived from the formal specifications.

Finally, many methods contain techniques for improving the performance of a system once built: **optimization** techniques. Once a system has been tested and found to function as required, optimizations reduce its size, cost, or resource consumption without changing its operation. Optimizations, however, sometimes make the structure of the system less obvious, for instance, by combining logically separate objects, propagating physical information across a logical interface, or reallocating functional units to balance processing requirements or minimize physical separations. When this is done, it is crucial that as much traceability as possible be retained through the transformations.

## 7.5. Summary

When software developers use a method, some tasks are almost wholly prescribed, others are guided, and still others are barely addressed. The use of a method must therefore judiciously combine strict adherence to rules, informed pursuit of guidelines, and tasteful innovation. A method typically helps by partitioning the problem into more manageable units, providing appropriate views of the system at each stage, and connecting units, views, and stages together in a manner that preserves traceability, supports analysis, and maintains a sound development framework.

As discussed previously, the assistance of specific tools can be invaluable in making methods easier to use, rules easier to follow, and development units easier to manage and control.

# 8. Choosing Methods

In the case of software products, the following must occur: "User needs and constraints must be determined and explicitly stated; the product must be designed to accommodate implementors, users and maintainers; the source code must be carefully implemented and thoroughly tested; and supporting documents . . . must be maintained" [Fairley 85]. In addition, the engineering process must deal with real-world constraints that limit the problem solution space and often limit the approaches that can be taken in the process of transforming the requirements into an acceptable problem solution.

Software development methods must deal with more than the process and techniques that translate the functional, performance, and other requirements of a software system into an operational system. They must deal with the environment that exists today in the area of computers and computing. Software engineers with diverse backgrounds and training are asked to solve problems with wider ranges of diversity, increasing complexity, and need for reliability. They must use varying implementation languages on diverse hardware and operating system platforms.

The process of choosing methods is one that must deal with the problems, goals, and situations alluded to above. Earlier work on methods [McDonald 85] indicates that method evaluation should be based on a three-step process: classification, evaluation, and selection. While this is true, it is important to remember the overall context of method selection. Methods are selected to aid the solution of particular problems in particular situations. The process of selecting methods in support of the engineering of software must be based on the following activities.

1. Analyzing and understanding (classifying) the characteristics of the engineering problem to be solved, the nature of the system to be built, the constraints on the permitted solution, and the process that must be followed.
2. Analyzing and understanding existing methods from various aspects: technical, process, usage.
3. Selecting a method based on its ability to help the engineers who will use it to deal with the problem, the solution constraints, and the engineering process.

The rest of this chapter discusses each of these activities in turn and gives an overview of the issues involved in selecting methods. Emphasis is placed on issues dealing with the engineering of real-time systems.

## 8.1. The Engineering Problem

The first area to consider when choosing a problem-solving method is the nature of the problem to be solved. Engineers who develop systems must deal with the characteristics of the system to be built as well as constraints that are placed on the software implementation. In addition, they must follow a development process that is defined by the organization in which they work. The following subsections discuss these areas in turn.

1. **System characteristics**. Software systems are called upon to support a wide variety of applications, and each system generally comes with its own unique set of

problems. Individual systems can generally be classified into one of several categories (transaction processing, business data processing, real-time, etc.) where members of each category share a common set of characteristics. An understanding of the nature of real-time systems, the characteristics they have in common with other system types, and the characteristics that set them apart is the first step in method selection.

Section 2.1 of this report lists characteristics that are unique to real-time systems. These characteristics are important to consider when choosing engineering methods for real-time systems in that the characteristics represent especially difficult problems that have received little support. Effective methods will be those that help the engineer deal with these characteristics as well as those that are common to all system types. General characteristics shared with other types of systems have been described in [McDonald 85].

2. **Implementation constraints.** The characteristics of real-time systems present the designers and implementors of typical systems with a set of difficult problems. Unfortunately, this is not the only set of problems that must be considered. In many situations, the solutions to the problems are constrained by the available technology and by the needs of organizations acquiring, developing, using, and maintaining systems.

   The task of system developers includes the activities of analysis, decomposition, and understanding of the systems operational requirements. It also includes analyzing and understanding the solution's constraints to compose a solution that operates within those constraints. Effective methods must support and preferably guide developers' efforts to compose an acceptable solution as they work in engineering organizations.

   Software solution constraints generally fall into four major areas. Detailed discussion of many of these characteristics can be found in [Fairley 85].

   - Hardware architecture constraints: These are generally imposed by existing hardware technology as it relates to the special environments in which the hardware must operate. Real-time systems have severe performance requirements; they need reliability in harsh environments and they need to sense and effect conditions in the external environment. These needs, coupled with the state of existing hardware technology, create hardware platforms with special characteristics. Software implemented on these platforms must account for the special characteristics.

   - Software architecture constraints: These are generally imposed by the need to cost-effectively create and maintain software that operates reliably on the constrained hardware. In addition, implementations are often constrained to use "standard" operating systems, language systems, and existing software components. These standards are often imposed with the goal of reducing the complexity, development time, and life-cycle cost of the system.

   - Integration and testing constraints: These are generally imposed by performance requirements, time pressures, logistics, and economics of typical development projects. Software integration and testing are often overlooked in discussions of development methods, but testing real-time systems is an especially difficult task that should be supported by development methods. Methods used to develop real-time systems must support the process of integration as well as test case generation, test execution, and analysis of test results. They should aid in extrapolating the results of tests executed on hosts, integration systems, and target system simulators to predict actual characteristics of the software on the real-target hardware. They shou[illegible]

the developers understand what testing can validly be done on host and integration systems and what areas cannot be adequately covered using this approach.

- Evolution constraints: These are typically imposed by the need to operate the system over long periods of time and to improve the system over its life cycle. Improvements can take the form of expanded functionality to meet new needs or the form of an enhanced or modified hardware platform. Hardware platforms are often modified to take advantage of the characteristics of newer hardware technology that provides components with characteristics such as greater reliability, decreased weight and size, or lower power consumption.

3. **Process constraints**. The software engineering activity occurs in the context of commercial organizations that build systems for profit. In addition, the systems are acquired by commercial or government organizations that need to solve their problems in the most cost-effective manner. The costs to consider are not only the costs of initially acquiring a system, but also the costs involved with operating and maintaining the system over long periods of time.

   The need to control cost in all its forms places the software engineering process — and those who conduct it — in a set of managed activities that are frequently reviewed and adjusted by those responsible for the cost of system production, system acquisition, and ownership.

   Section 4.5 discusses the engineering process and the process-related constraints that are placed on the engineering activities. The point to recognize here is that the most effective methods are those that deal with management issues as well as technical issues, and do so by understanding the work products needed to support the technical activities.

Recognizing the above, it is apparent that software developers are not free to create any software structures, constructs, or mechanisms they can imagine. Each implementation platform (hardware and operating/language system) forces developers in particular directions. The desire to reuse existing software components limits the developers even more. The most effective methods are those that deal with the implementation constraints. The methods should provide assistance in accurately analyzing and describing the system to be built and should provide a bridge between the system description and an acceptable implementation architecture. The method should lead directly into an implementation made up of the entities, constructs, and mechanisms supported by the implementation platform. The method should assist in defining a cost-effective testing process. Finally, the method should support the management practices and engineering process constraints that are generally found in the engineering process. Identifying and classifying the characteristics of the engineering problem help the organization identify the needs a method must address.

## 8.2. Classifying Methods

The previous section outlined the first step necessary in choosing methods: characterization of the engineering problem to be solved in terms of the nature of the system to be built, the constraints placed on the implementation, and the constraints placed on the engineering process. The second step in the process of choosing a method deals with the characterization of existing methods.

Chapter 5 outlined methods as they have traditionally been described. These descriptions and characterizations are useful for understanding what a method is and how it addresses particular problems, but they are not always helpful for deciding which methods apply to particular situations. Useful characterizations of methods should be made in terms of their relationship to the characteristics of the system to be built, the constrained solution, and the process to be followed when using the method.

Earlier work in methods ( [Freeman 83] and [McDonald 85]) collectively define five categories of characteristics to be used in classifying methods: technical, management, economic, usage, and Ada compatibility. While we agree that these are five useful categories, for the purpose of this report — a high-level introduction to methods and their selection for use — we have combined Ada compatibility with technical characteristics, and economic issues with management issues. The following three subsections deal with classifying methods in terms of their technical, management, and usage characteristics. The process of "characterizing" concerns itself with what a method is, what a method does, and what issues a method addresses.

1. **Technical characteristics**. Chapter 3 discussed various aspects of the development process. It outlined three stages of development (specification, design, and implementation) and three views of the system being developed (behavioral, functional, and structural). This high-level view of the process can be used as a framework for classifying the technical characteristics of a method.

   The characteristics of the engineering problem to be solved are dealt with during the specification stage. These characteristics can be grouped into the behavioral and functional views appropriate for the specification stage. These views of the problem are carried forward through the design to the implementation stage where they are represented as the behavioral and functional views of the implementation. Effective methods are those that allow smooth transition across stages and trace the functional and behavioral aspects of the problem through to implementation.

   The solution constraints are dealt with beginning at the design stage and moving forward to the implementation stage. At the design stage, the behavioral and functional characteristics of the problem are mapped to the behavioral and functional characteristics of the solution. The structural view prescribed by an effective method is one that maps directly to the functional/structural view of the implementation stage.

2. **Management characteristics**. As stated in Section 5.3, the most effective methods are those that deal with management and technical issues. When characterizing a method, it is important to consider the support it gives to management issues. The characterization should consider whether and how the method deals with the typical management and project issues of estimating, planning, review, and management of work products. This characterization should be related to the needs and process that exist within the organization. Management practices are often the most difficult thing to change within an organization. Relating a method to existing practices will help the selection process by identifying potential changes in practice required to use the method effectively.

3. **Usage characteristics**. When classifying methods for comparison and evaluation, it is important to capture and describe the characteristics of the method that will affect its use by the engineering organization. Usage characteristics include the conceptual basis for the method, availability of training, and automated support. Again, this characterization should be related to the organization that will use the method. Selecting a new method often means a change in engineering practice.

The classification and selection process should help the organization understand the magnitude of the change, and help engineers to make the change.

Characterizing existing methods is, then, the second step in the selection process. The classification should be done in a manner that allows those responsible for method selection and use to relate the methods to the needs of the organization as it develops specific systems.

## 8.3. Selecting a Method

The previous sections discussed the first two steps of the selection process: identifying the needs of the organization and classifying existing methods in terms of technical, management, and usage characteristics. These first steps set the stage for the last by helping those responsible for the selection come to grips with the nature of the problem they are trying to solve and the potential of specific methods to deal with the problem.

The final step is to select a method, though this is not usually a simple procedure. The needs analysis performed as the first step of the selection process will generally produce a set of needs that are not completely covered by any method. The process of selection is one that finds the best solution rather than a complete solution. The following general activities should be performed in making the selection.

1. **Rank the issues.** The first step of the selection process results in a set of issues that must be considered by those producing a particular system. This set of issues must be ranked by the engineering organization to determine its real needs. The ranking should be based on the importance of each issue to the particular organization and should consider questions such as: How critical is resolution of the issue to the success of the engineering project? How difficult will the issue be to resolve? How much experience and success has the organization had in dealing with similar problems in the past? The ranking should identify problems that seem especially difficult to solve or those that will require the organization to use unfamiliar techniques. This ranked list should cover both technical and management issues. Once created, it can serve as the basis for the step described below.

2. **Identify key discriminators.** The ranked list of issues can be used to define the characteristics of a method that will be effective in dealing with the engineering problem. The ranked list should be used to define two classes of features: essential features and desirable features. While the list of desirable features will probably contain items that address all the issues, it is important to remember that one is unlikely to find any method that will do this. Desirable features are those where there is room for compromise — those that can possibly be absent or only partially effective. Essential features, on the other hand, are those where there is no room for compromise. This list of key discriminating factors can be used to quickly weed out candidate methods.

3. **Apply to methods.** Once the key discriminators are identified, they should be applied to the classified methods to determine which methods have potential benefit. The result of this process will usually be a shorter list of candidate methods. The candidates are those that possess all essential features — features that deal effectively with the critical issues defined in the ranking process. Further evaluation should consider how well each method covers the desirable features. This is generally a subjective process that must deal with issues not easily measured and do so in a way that considers the culture of the engineering organization.

4. **Determine undesirable features.** During the final steps of the selection process, it is important to consider the undesirable features of the candidate methods. Methods have been developed to deal with the complexities of engineering activities, and each handles complexity in its own way. Rigorous, formal specification methods, for example, lead to problem statements that are complete and consistent. These methods, however, are often difficult to learn and cumbersome to use. Certain other graphically based methods, on the other hand, are relatively easy to use and lead to problem descriptions that can be understood and reviewed by wide audiences. These methods, however, often lack the rigor and formal basis required to support detailed, thorough analysis for a variety of conditions.

   For each method evaluated, it is important to determine the method's undesirable features. What issues does the method not deal with? What are the negative consequences of using the method to solve a particular problem? What costs are associated with acquiring and adopting the method?

The set of activities described above brings those responsible for method selection to a decision point. From the preceding discussion, it should be apparent that the decision is complicated and largely subjective. It involves a number of tradeoffs, for example, rigor versus ease of use, management support versus minimal engineering workload. In many cases, the tradeoff decisions cannot be supported with well-defined, objective measurement criteria since these metrics do not yet exist.

The final decision is best attained by thoroughly analyzing and understanding the engineering problem, the constrained solution, and the candidate methods. This understanding, used by those experienced in the development of real-time systems, supports the subjective decision-making process that must occur. The decision-making process should identify the best method for the problem. It should also identify those issues the method does not deal with and the costs involved in adopting the method. The data collected and the understanding achieved during the selection process can be used to support the successful adoption of the method selected.

# Appendix A: Key Methods - Acronym Descriptions and References

| Method | Brief Description | References |
|---|---|---|
| A7 | A7 Methodology | [Parnas 86] |
| DARTS | Design Approach for Real-Time Systems | [Gomaa 86] |
| DCDS | Distributed Computing Design System | [Alford 85] |
| HIPO | Hierarchy, plus Input, Process, Output | [Stay 76] |
| HOS | Higher Order Software | [Hamilton 76] |
| JSD | Jackson Structured Design | [Cameron 83]<br>[Jackson 75] |
| MASCOT | Modular Approach to Software Construction Operation and Test | [Bate 86]<br>[Booer 82]<br>[Jackson 84]<br>[Simpson 84] |
| OOD | Object-Oriented Design | [Booch 83b]<br>[Buzzard 85]<br>[Cox 84] |
| PAISLey | Process-Oriented, Applicative, and Interpretable Specification Language | [Zave 86] |
| PAMELA | Process Abstraction Method for Embedded Large Applications | [Cherry 87] |
| PSL/PSA | Problem Statement Language/ Problem Statement Analyzer | [Teichrow 77] |
| SADT | Structured Analysis and Design Technique | [Ross 77]<br>[Connor 80]<br>[Ross 81] |
| SARA | System Architect's Apprentice | [Estrin 86] |
| SA/SD/RT | Structured Analysis/ Structured Design/ Real-Time | [DeMarco 78]<br>[Ward 85] |
| SREM | Software Requirements Engineering Methods | [Alford 85]<br>[Hoffman 81]<br>[Scheffer 85] |
| Statecharts | Statecharts | [Harel 86] |
| VDM | Vienna Development Method | [Bjorner 83]<br>[Jones 86] |
| Warnier Orr | Warnier Orr Diagrams | [Warnier 76] |

## References


[Aho 83] Aho, Alfred V.; Hopcroft, J. E.; and Ullman, J. D.
*Data Structures and Algorithms.*
Addison-Wesley, Reading, MA, 1983.

[Alford 85] Alford, Mack.
SREM at the Age of Eight: The Distributed Computing Design System.
*Computer* 18(4):36-46, April, 1985.

[Archer 86] Archer, J. E., and Devlin, M. T.
Rational's Experience Using Ada for Very Large Systems.
In *Proceedings of First International Conference on Ada Programming Language Applications for the NASA Space Station*, pages B.2.5.1-12. NASA, June, 1986.

[Balzer 85] Balzer, R.
A 15 Year Perspective on Automatic Programming.
*IEEE Transactions on Software Engineering* SE-11(11), November, 1985.

[Bate 86] Bate, G.
MASCOT Overview.
*IEEE, London* , January, 1986.

[Bergland 81] Bergland, G.D.
A Guided Tour of Program Design Methodologies.
*Computer* 14(10):13-37, October, 1981.

[Bjorner 82] Bjorner, D., and Jones, C. B.
*Formal Specification and Software Development.*
Prentice/Hall International, London, England, 1982.

[Bjorner 83] Bjorner, D., and Prehn, S.
*Software Engineering Aspects of VDM, The Vienna Development Method.*
North-Holland, Amsterdam, Netherlands, 83.

[Boehm 86] Boehm, B. W.
A Spiral Model of Software Development and Enhancement.
*ACM SIGSOFT Software Engineering Notes* 11(4), August, 1986.

[Booch 83a] Freeman, P., and Wasserman, A. (editors).
*Object-Oriented Design.*
In Tutorial: Software Design Techniques, IEEE Computer Society Press, Washington, DC, 1983.

[Booch 83b] Booch, Grady.
*Software Engineering with Ada.*
Benjamin/Cummings, Menlo Park, CA, 1983.

[Booer 82] Booer, A. K., and John, W. T.
MASCOT Real-Time Software Development Using the Context[illegible]n.
*Colloquium on Development Environments for Microprocessor Systems* , May, 1982.

[Brooks 87] Brooks, F. P.
The Silver Bullet, Essence and Accidents of Software Engineering.
*IEEE Computer* 20(4), April, 1987.

[Buzzard 85] Buzzard, G. D., and Mudge, T. N.
Object-Based Computing and the Ada Language.
*Computer* 18(3):11-19, March, 1985.

[Cameron 83] Cameron, J.
*Tutorial: JSP and JSD: The Jackson Approach to Software Development.*
IEEE Computer Society Press, Washington, DC, 1983.

[Cherry 87] George W. Cherry.
*Introduction to PAL and Pamela II (Process Abstraction Language and Process Abstraction Method for Embedded Large Applications).*
P.O. Box 2429, Reston, VA 22090, 1987.

[Connor 80] Connor, Michael F.
Structured Analysis and Design Technique (SADT) Introduction.
*IEEE Engineering Management Conference Record* :138-143, May, 1980.

[Cox 84] Cox, B. J.
Message/Object Programming: An Evolutionary Change in Programming Technology.
*IEEE Software* 1(1):50-62, January, 1984.

[Dart 87] Dart, Susan A.; Ellison, Robert J.; Feiler, Peter H.; and Habermann, N. A.
Trends in Software Development Environments.
Carnegie Mellon University, Pittsburgh, PA.
May, 1987

[Date 86] IBM Editorial Board (editor).
*An Introduction to Database Systems.*
Addison-Wesley, Menlo Park, CA, 1986.

[DeMarco 78] DeMarco, Tom.
*Structured Analysis and System Specification.*
Yourdon, Inc., New York, 1978.

[DoD 82] Ada Joint Program Office.
*Ada Methodologies: Concepts and Requirements.*
Technical Report, Department of Defense, November, 1982.

[Estrin 86] Estrin, G.; Fenchel, R.S.; Razouk, R.R.; and Vernon, M.K.
SARA (System Architects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems.
*IEEE Transactions Software Engineering* SE-12(2):293-311, February, 1986.

[Fairley 85] Fairley, Richard E.
*Software Engineering Concepts.*
McGraw Hill, New York, 1985.

[Firesmith 87] Firesmith, Donald G.
Software Development Process.
*Defense Science and Electronics* 6(7):56-59, July, 1987.

[Firth 87] Firth, Robert; Mosley, Vicky; Pethia, Richard; Roberts, Lauren; and Wood, William.
*A Guide to the Classification and Assessment of Software Engineering Tools*
Carnegie Mellon University, Pittsburgh, PA, 1987.

[Floyd 86] Floyd, Christiane.
A Comparative Evaluation of System Development Methods.
In Olle, T.W.; Sol, H. G.; and Verrijn-Stuart, A. A. (editors), *Information Systems Design Methodologies: Improving the Practice*. North-Holland, 1986.

[Freeman 83] Freeman, P., and Wasserman, A. I.
Ada Methodologies: Concepts and Requirements.
*ACM SIGSOFT Software Engineering Notes* , 1983.

[Gallo 87] Gallo, F.; Minot, R.; and Thomas, I.
The Object Management System of PCTE as a Software Engineering Database Management System.
In *Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 12-15. ACM, January, 1987.

[Goldberg 84] Goldberg, Adele.
*Smalltalk-80, The Interactive Programming Environment.*
Addison-Wesley, Reading, MA, 1984.

[Gomaa 86] Gomaa, H.
Software Development of Real-Time Systems.
*Communications ACM* 29(7):657-8, July, 1986.

[Guttag 85] Guttag, J. V.; Horning, J. J.; and Wing, J. M.
*Larch in Five Easy Pieces.*
Technical Report, Digital Systems Research Center, July, 1985.

[Habermann 86] Habermann, A. N., and Notkin, D.
Gandalf: Software Development Environments.
*IEEE Transactions on Software Engineering* SE-12(12):1117-1127, December, 1986.

[Hamilton 76] Hamilton, M., and Zeldin, S.
Higher Order Software - A Methodology for Defining Software.
*IEEE Transactions on Software Engineering* SE-2(1):9-32, 1976.

[Harel 86] Harel, David.
Statecharts: A Visual Approach to Complex Systems.
*Concurrent Systems* , February, 1986.

[Hayes 87] Hayes, I.
*Specification Case Studies.*
Prentice/Hall International, London, England, 1987.

[Hesse 84] Hesse, Wolfgang.
A Systematics of Software Engineering: Structure, Terminology and Classification of Techniques.
NATO ASI Series: Program Transformation and Programming Environments, Vol F8.
1984

[Hoare 85] Hoare, C. A. R.
*Communicating Sequential Processes.*
Prentice/Hall International, London, England, 1985.

[Hoffman 81] Hoffman, R. H.; Loshbough, R. P.; and Smith, R. W.
SREM: Software Requirements Engineering Methodology and the Requirements Engineering and Validation System (REVS).
In *Proceedings of the NBS/IEEE/ACM Software Tool Fair*, pages 111-16. IEEE, ACM, NBS, Washington, DC, San Diego, CA, March, 1981.

[Houghton 87] Houghton, Raymond C., and Wallace, D. R.
Characteristics and Functions of Software Engineering Environments: An Overview.
*ACM SIFSOFT Software Engineering Notes* :64-84, January, 1987.

[Jackson 75] Jackson, Michael A.
*Principles of Program Design.*
Academic Press, New York, 1975.

[Jackson 84] Jackson, K.
MASCOT.
*IEEE Colloquium on MASCOT 3 (Digest No. 113)* , December, 1984.

[Jones 86] Jones, C. B.
*Systematic Software Development Using VDM.*
Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Kelly 87] Kelly, John C.
A Comparison of Four Design Methods for Real-Time Systems.
Research paper supported by a NASA/ASEE fellowship at JPL, CA.
1987

[Leblang 85] Leblang, D. B., and McLean, G. B.
Configuration Management for Large-Scale Software Development.
In *GTE Workshop on Software Engineering Environments for Programming in the Large.* June, 1985.

[Lehman 85] Lehman, M. M.
Approach to a Disciplined Development Process - The ISTAR Integrated Project Support Environment.
Imperial Software Technology, Ltd.
November, 1985

[Leveson 83] Leveson, N. G., and Harvey, P. R.
Analyzing Software Safety.
*IEEE Transactions on Software Engineering* SE-9(5):569-579, September, 1983.

[Love 83] Love, Tom.
Experiences with Smalltalk-80 for Application Development.
*Proceedings of SoftFair (IEEE Order No. 83CH1919-0)* , July, 1983.

[McDonald 85] McDonald, Catherine W.; Riddle, William; and Youngblut, Christine.
STARS Methodology Area Summary - Vol II: Preliminary Views on the Software Life Cycle and Methodology Selection.
Prepared for Office of the Undersecretary of Defense for Research and Engineering, IDA Paper P-1814.
March, 1985

[MIL-STD-CAIS 85]
Ada Joint Program Office.
*Military Standard Common APSE Interface Set.*
Ada Joint Program Office, Washington, DC, 1985.

[Parnas 72] Parnas, D. L.
On the Criteria To Be Used in Decomposing Systems Into Modules.
*Communications ACM* 15(12):1053-1058, December, 1972.

[Parnas 86] Parnas, D. L., and Clements, P. C.
A Rational Design Process: How and Why To Fake It.
*IEEE Transactions on Software Engineering* SE-12(2), February, 1986.

[Partsch 83] Partsch, H., and Steingruggen, R.
Program Transformation Systems.
*ACM Computing Surveys* 15(3), September, 1983.

[PavePillar 85] AFWAL/AAAS-1.
Architecture Specification for Pave Pillar Avionics, SPA 900-99001.
June, 1985

[Raghavan 86] Raghavan, Sridhar, A., and Chand, Donald, R.
*Applications Generators and Fourth Generation Languages.*
Technical Report TR-86-02, Wang Institute, February, 1986.

[Ross 77] Ross, D.T., and Schoman, K.E.
Structured Analysis for Requirements Definition.
*IEEE Transactions Software Engineering* SE-3(1):69-84, January, 1977.

[Ross 81] Bergland, G. D., and Gordon, R. D. (editors).
*Tutorial on Software Design Techniques: Structured Analysis (SA): A Language for Communicating Ideas.*
IEEE Computer Society Press, Los Angeles, CA, 1981.

[Royce 70] Royce, W. W.
Managing the Development of Large Software Systems: Concept and Technique.
In *Proceedings of the 9th International Conference on Software Engineering.* August, 1970.

[Scheffer 85] Scheffer, Paul A., and Stone, Albert H. III.
A Case Study of SREM.
*Computer* 18(4):47-54, April, 1985.

[Simpson 84] Simpson, H. R.
MASCOT 3 (Real-Time Software Design Methodology).
*IEEE Colloquium on MASCOT 3 (Digest No. 113)* , December, 1984.

[Smith 85] Smith, Douglas R.; Kotik, G. B.; and Westfold, S. J.
Research on Knowledge-Based Software Environments at Kestrel Institute.
*IEEE Transactions on Software Engineering* SE-11(11), November, 1985.

[Stay 76] Stay, J F.
HIPO and Integrated Program Design.
*IBM Syst Journal* 15(2):143-154, 1976.

[Teichrow 77] Teichrow, Daniel.
PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.
*Transaction Software Engineering* SE-3(1), January, 1977.

[Teitelbaum 81] Teitelbaum, T.; Reps, T.; and Horwitz, S.
The Why and Wherefore of the Cornell Program Synthesizer.
*IEEE Tutorial on Software Development Environments* :64-72, 1981.

[Ward 85] Ward, Paul T., and Mellor, Stephen J.
*Structured Development for Real-Time Systems, Vol I: Introduction & Tools.*
Yourdon Press, New York, 1985.

[Ward 86] Ward, Paul T.
The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing.
*IEEE Transactions on Software Engineering* SE-12(2), February, 1986.

[Warnier 76] Warnier, J.D.
*Logical Construction of Programs.*
VanNostrand Reinhold, New York, 1976.

[Yau 86] Yau, Stephen S., and Tsai, Jeffery J. P.
A Survey of Software Design Techniques.
*IEEE Transactions on Software Engineering* SE-12(6):713-721, June, 1986.

[Zave 86] Zave, Pamela.
Salient Features of an Executable Specification Language and Its Environment.
*IEEE Transactions on Software Engineering* , February, 1986.

## REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS<br>NONE |
| 2a. SECURITY CLASSIFICATION AUTHORITY<br>N/A | 3. DISTRIBUTION/AVAILABILITY OF REPORT<br>APPROVED FOR PUBLIC RELEASE<br>DISTRIBUTION UNLIMITED |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE<br>N/A | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>CMU/SEI-87-TR-41 | 5. MONITORING ORGANIZATION REPORT NUMBER(S)<br>ESD-TR-87-204 |
| 6a. NAME OF PERFORMING ORGANIZATION<br>SOFTWARE ENGINEERING INSTITUTE<br>6b. OFFICE SYMBOL (If applicable)<br>SEI | 7a. NAME OF MONITORING ORGANIZATION<br>SEI JOINT PROGRAM OFFICE |
| 6c. ADDRESS (City, State and ZIP Code)<br>CARNEGIE MELLON UNIVERSITY<br>PITTSBURGH, PA 15213 | 7b. ADDRESS (City, State and ZIP Code)<br>ESD/XRS1<br>HANSCOM AIR FORCE BASE, MA 01731 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION<br>SEI JOINT PROGRAM OFFICE<br>8b. OFFICE SYMBOL (If applicable)<br>SEI JPO | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>F1962885C0003 |
| 8c. ADDRESS (City, State and ZIP Code)<br>CARNEGIE MELLON UNIVERSITY<br>SOFTWARE ENGINEERING INSTITUTE JPO<br>PITTSBURGH, PA 15213 | 10. SOURCE OF FUNDING NOS.<br>PROGRAM ELEMENT NO.: —<br>PROJECT NO.: N/A<br>TASK NO.: N/A<br>WORK UNIT NO.: N/A |
| 11. TITLE (Include Security Classification)<br>A CLASSIFICATION SCHEME FOR SOFTWARE DEVELOPMENT METHODS | |
| 12. PERSONAL AUTHOR(S)<br>FIRTH, WOOD, PETHIA, ROBERTS, MOSLEY, DOLCE | |
| 13a. TYPE OF REPORT<br>FINAL<br>13b. TIME COVERED<br>FROM _____ TO _____ | 14. DATE OF REPORT (Yr., Mo., Day)<br>NOVEMBER 87<br>15. PAGE COUNT<br>58 |
| 16. SUPPLEMENTARY NOTATION | |
| 17. COSATI CODES<br>FIELD / GROUP / SUB GR. | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)<br>SOFTWARE ENGINEERING, SOFTWARE DEVELOPMENT, REAL-TIME |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

SOFTWARE DEVELOPMENT METHODS ARE USED TO ASSIST WITH THE PROCESS OF DESIGNING SOFTWARE FOR REAL-TIME SYSTEMS. MANY SUCH METHODS HAVE COME INTO PRACTICE OVER THE LAST DECADE, AND NEW METHODS ARE EMERGING. THESE NEW METHODS ARE MORE POWERFUL THAN THE OLD ONES, ESPECIALLY WITH REGARD TO REAL-TIME ASPECTS OF THE SOFTWARE. THIS REPORT DESCRIBES A CLASSIFICATION SCHEME FOR SOFTWARE DEVELOPMENT METHODS, INCLUDES DESCRIPTIONS OF THE MAJOR CHARACTERISTICS OF SUCH METHODS, AND CONTAINS SOME WORDS OF ADVICE ON CHOOSING AND APPLYING SUCH METHODS.

| | |
|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT<br>UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒ | 21. ABSTRACT SECURITY CLASSIFICATION<br>UNCLASSIFIED, UNLIMITED |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL<br>KARL SHINGLER | 22b TELEPHONE NUMBER (Include Area Code)<br>(412) 268-7630<br>22c. OFFICE SYMBOL<br>SEI JPO |

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.